GENERAL PURPOSE SIMULATOR FOR MULTICOMPUTER ARCHITECTURES

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the degree of

MASTER OF TECHNOLOGY

by

Jayashree M. Yabannavar

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JULY 1992

CERTIFICATE

It is certified that the work contained in the thesis **General purpose simulator for multicomputer architectures**, by **Jayashree M. Yabannavar** has been carried out under my supervision and this work has not been submitted elsewhere for a degree.

Dr. Rajat Moona

Dept. of Computer Science and engineering

July 1992

ACKNOWLEDGEMENTS

I am extremely indebted to Dr. Rajat Moona my thesis supervisor, for the valuable guidance he gave me at every stage of the preparation of thesis. Working with him, in very cordial and yet professional atmosphere he created, has been a special and memorable experience for me.

I am highly obliged to Mr. Deepak Gupta who helped me to carry out my thesis work smoothly. I am thankful to Shylaja.S.V and J.J. Hazel Das who helped me in document preparation.

My thanks to all faculty members of Dept. of Computer Science and Engg. IITK, who with their able guidance made my pursuit of M.Tech. here, an intellectually rewarding exercise. Thanks also to the staff of the department who helped in various ways.

I specially thank Anilkumar and Kum. Anup, who were the source of inspiration through the course of M.Tech degree. I thank all my friends who have made my stay at IITK a pleasant and memorable one.

JAYASHREE.

ABSTRACT

Multicomputers are message passing based multiprocessor systems. Here processing units operate asynchronously under the control of local controller, one for each processing element. Hence a problem that has an arbitrarily structured parallelism, can be programmed with much flexibility on multicomputers. In this thesis, a general purpose simulator is developed with the motivation of providing a test bed for developing and testing concurrent algorithms for multicomputer architectures. It is implemented in three layers. Process creation and interprocessor communication to simulate single processing element is implemented in the first layer, the second layer which is built over the first is specific to the particular class of multicomputers and provides better user interface. User program is implemented calling the primitives provided in the second layer. This package provides facilities to simulate both, point to point and broadcast communication multicomputer architectures.

TABLE OF CONTENTS

List of Figures

CHAPTER 1: INTRODUCTION

Multiple processor systems are now being increasingly used for high speed computations. They can be broadly [redacted] divided into two categories - the shared memory systems (multiprocessors) and the message passing based systems (multicomputers). In the message passing type of multicomputer system, processing units operate asynchronously under the control of a local controller one for each processing unit[Reed87]. A problem that has an arbitrarily structured parallelism, can be programmed with much flexibility on a multicomputer system. An effort in the direction of developing the parallel programming environment for multicomputer systems lead us to the development of the package, a general purpose simulator for multicomputer architectures. This simulator can be used either in the multiple program multiple data mode or in the single program multiple data mode depending on whether the functions to be executed by the processors are same or not.

1.1 MOTIVATION

The motivation for this work is to build a parallel programming environment for simulating multicomputer architectures. We do not have knowledge of any implementation which have a comprehensive approach to simulation. More precisely, the implementations thus far, have been targeted for specific multicomputer architectures and cannot be easily adapted to simulate any parallel machine. What we have in mind is to build a general purpose

this simulator provides a comprehensive platform for parallel processing which would encourage experimentation with parallel algorithms in various application areas.

1.2 CONTRIBUTIONS AND OUTLINE OF THESIS

In this thesis, a general purpose simulator tool, for multicomputers is built which can be dynamically reconfigured, to simulate any user defined network of multicomputers.

A multicomputer can be characterised by a number of processing elements and a set of data routing functions provided by the interconnection network[Siegel79]. The processing elements are independent of the network topology, and therefore some general purpose routines are designed and implemented in the first stage, namely simulator core. In order to provide a mechanism for interprocessor communication for both point to point and broadcast communication, modules are also implemented in this core.

We have maintained a structure interface for each topology, wherein a set of interconnection networks can be predefined in structure core. In order to facilitate this a number of auxiliary routines are provided in the simulator core. Apart from the interconnection network the structure core also provides an interface to the interprocess communication in accordance with the network defined.

With the primitives provided by the structure interface, user can easily write parallel programs and test his parallel algorithms.

1.3 THESIS ORGANISATION

The thesis is organised into six chapters including the present chapter. The concepts of process creation, interprocess communication, the structure of simulator and structure cores are introduced in chapter 2. Chapter 3 discusses the implementation of simulator. Chapter 4 details few examples of interfaces to simulator core. Chapter 5 presents the simulation runs, and few examples of user programs. Conclusions and scope for further work are outlined in chapter 6.

Appendix A contains the routines available for structure core. Appendix B illustrates few examples of structure file. Finally, Appendix C explains how to run user's program using the simulator.

CHAPTER 2 : DESIGN OF SIMULATOR

2.1 INTRODUCTION

The simulator can be organised into three modules. In the first module called simulator core, processing element simulation is developed, which has been dealt with in section 2.2. In the second module, structure core, an interface to the previous module is designed to encapsulate all configurations of a particular class of multicomputers. The structure core is discussed in section 2.3. The third module involves the development of user program with the primitives provided by the structure core. In section 2.4 we discuss the third module. Finally we conclude in section 2.5.

2.2 SIMULATOR CORE

To simulate single processing element of the network, the simulator package has to create a process. Therefore, a module for process creation is developed which consists of a number of routines. Each newly created process is the exact replica of the creating process, since all the processors have the same status in the multicomputer network.

The asynchronous model of communication is chosen which is the one generally used in multicomputers. This is also a more natural model for the programmer. A synchronous model of communication can always be built over the asynchronous one in the structure core. The key issues involved in interprocessor communication of both point to point and

1. Communication links

2. Design of communication routines.

2.2.1 Communication links

The interprocessor links for point to point communication networks, can be simulated using named pipes[Bach86], unnamed pipes and sockets. The socket mechanism is not chosen in our approach because it adheres to a server-client model which is inherently different from the type of communication in multicomputers wherein all processors have equal status. For using unnamed pipes, the parent process must open the pipe before creating another process so that the child process can share it. As every process uses two pipes for bi-directional communication, this approach exceeds the operating system defined upper limit of the number of open files per process, even for very small multicomputer configurations.

The advantage of using named pipes is that, it is not passed to child process using parent_child inheritance. Thus, if we have a pipe naming protocol which gives the name of the pipe used for communication between the processors, communication can be established and no process will have to open pipes more than twice the number of ports per processor, for bi-directional communication.

However, this approach cannot be used for broadcast communication because pipes provide mechanism of communication only between two processors. Hence we use files with supervisory locks, one for each broadcast bus to solve the problem of inconsistency. It facilitates the

after doing the job by unlocking it to the other processors which are connected on the same bus.

2.2.2 Design of Communication Routines

Simulator uses blocked mode of communication for receiving message and unblocked mode of communication for sending message. Thus the processor receiving a message waits for the message to arrive if it has not arrived already. But the sender of the message waits only in case the message buffer is full. This wait can be minimised by suitable choice of buffer size. Message length is kept as a variable. Thus the routine used for sending messages requires message length as an argument whereas the routine used for receiving messages returns the message length for similar reason.

2.2.3 Auxiliary Routines

A number of routines are developed which facilitate writing the structure core for any topology of multicomputer configuration. These are discussed in chapter 3.

2.3 STRUCTURE CORE

Design of the structure core depends on how simulator works so as to adapt any configuration of multicomputers. At the start of simulation, the control is given to the structure core. The structure core at this point takes the architecture dependent parameters as the input and establishes the interconnection network. It then initiates the simulator core. The simulator core sets up the simulation

between them taking input from structure core. It then passes the control to the user program. After doing the simulation, the simulation environment is disposed by terminating the processes and deleting the communication links. To incorporate such software protocol, we summarise the structure of the structure core as given below:

* an entry point to construct the interconnection network of multicomputers
* input interface
* an entry point to set the simulation environment is called
* an entry point to the user's program is called
* a protocol to delete the communication links is called
* an interface to the interprocess communication for the network is defined
* output interface

More about structure core is discussed in chapter 3. Some examples of structure core are discussed in chapter 4.

2.4 USER PROGRAM

User, in his parallel programs invokes the simulator using the primitive for process creation given in the structure core. In order to communicate the messages between the processes the protocol provided by the communication interface in the structure core, can be used by the user program. Examples of user program can be found in chapter 5, Simulation Runs.

2.5 CONCLUSION

In this chapter we described the different modules of general purpose simulator for multicomputers. It can be used to simulate variety of the algorithms designed for these architectures.

In the later chapters, we will be describing the implementation of simulator with few examples of structure core and some example user programs.

CHAPTER 3 : IMPLEMENTATION OF SIMULATOR

3.1 INTRODUCTION

The implementation of simulator can be broadly divided into three stages, namely simulator core, structure core and user program. In the simulator core, the general purpose routines for process creation and mechanisms for inter process communication are implemented. This core contains parts of simulator, specific to operating system and common to all configurations of multicomputers. In section 3.2 we discuss the simulator core implementation. In the second stage,using these general purpose routines, the primitives to simulate any given multicomputer configuration are implemented. The structure core is therefore specific to a processor topology. In general it is parameterised and can be used to encapsulate all configurations of a particular class of multicomputers. The structure core is discussed in section 3.3. The primitives provided by the structure core are used to develop the user program. Finally we conclude this chapter in section 3.4.

3.2 SIMULATOR CORE

The basic simulation routines of process creation, termination and raw mode of communication between the processor and its immediate neighbors in case of point to point communication, and the processors connected on the same bus in the case of broad_cast communication are implemented here.

3.2.1 Process Creation

Creation of process and generation of associated links for inter process communication is implemented through a number of routines in simulator core. Let us start with the routine to create the process, namely pfork().

PFORK subroutine

```
int pfork (i);
int i;
```

This subroutine creates the new process. The new process is an exact copy of the creating process. The newly created process creates and opens the named pipes as communication links to its immediate neighbors in case of point to point communication or create shared files for each bus in case of broad cast communication. If successful, this routine returns 0 or else it returns ERR_FORK killing all the processes created until now.

The procedure call create_pipes() called by each newly created process causes the creation of named pipes, as unidirectional communication links to its neighbors. For implementing bi-directional communication links two pipes are used. The pid of the parent is used in the name of the pipe, so that more than one simulations active at the same time don't have duplicate pipe names. If successful, the routine returns 0 or else returns ERR_PIPECREATE.

The procedure call open_pipes() called by the process opens the named pipes to its neighbors for communication. This procedure returns 0 in case of

The opening of pipes is not as simple it seems at first sight. This is because of the deadlock avoidance scheme in unix for pipes. The problem arises if a process opens a pipe for just reading, it is made to wait till another process opens the same pipe for writing and viceversa[Bach86]. So, if each process follows naive algorithm given in fig 3.2 for opening the pipes we can have classical deadlock situation in the following scenario.

Consider the hypercube in dimension 2. The processors are numbered as shown in fig 3.1. The following sequence of events is possible(even likely to occur):

i) The processor 0 tries to open the pipe from processor 1 for reading and gets blocked waiting for processor 1 to open it for writing.

ii) The processor 1 tries to open the pipe from processor 0 for reading and gets blocked waiting for processor 0 to open it for writing.

iii) The processor 2 tries to open the pipe from processor 0 for reading and gets blocked waiting for processor 0 to open it for writing.

iv) The processor 3 tries to open the pipe from processor 1 for reading and gets blocked waiting for processor 1 to open it for writing.

Processors 0,1,2,3 are now in classic deadlock situation. Similar is the fate of processors in higher dimensions also.

A simple solution to this problem would make each process open pipes for reading and writing even though it might use it for reading or writing only. In this case no process has to wait for another. But in unix even this solution is unfeasible because, whenever the process closes the pipe the data in the pipe is flushed if there are no more readers left[Bach86]. Thus the following sequence of events is possible:

i) Process 2 opens the pipe to process 0.

ii) Process 2 sends the message to process 0.

iii) Process 2 closes this pipe and exits.

iv) Process 0 opens this pipe and tries to read the message from process 2. Since the pipe has no data , process 0 waits forever for some message to arrive in the pipe.

The solution adopted in this approach is to have processes open the pipes in different order, in accordance with a protocol which ensures that a deadlock situation never arises.

According to this protocol:

1. Process 0 opens the pipe from process 1 and tries to read, it will be successful since process 1 opens the same pipe for writing first.

2. Process 1 opens pipe from 3 for reading and 3 opens the same pipe for writing first and hence process 1 succeeds to read the data from the pipe.

Similar is the case with the other processes. Fig 3.3 shows this protocol.

The complimentary routine of process creation is to terminate a process is called by a process when its execution is complete.

```
TERMINATE subroutine
void terminate();
```

This routine when called by a process causes the normal termination of the process.

After simulation, the simulation environment is disposed by terminating the processes and deleting the communication links. To delete all the files created during simulation, the subroutine clean is called by the process 0 before termination.

```
CLEAN  subroutine
void clean();
```

This routine when called by the process removes all the named pipes and shared files (if created).

3.2.2 Interprocessor Communication

Having described the creation and termination of processes, we now describe communication routines which are called by the processing elements in a multicomputer configuration for communicating among themselves. The communication routines are implemented by the complementary pairs of subroutines namely read_from and write_to.

WRITE_TO subroutine

```
int write_to (which, mesg, len)
int which, len;
char *mesg;
```

The routine write_to writes 'len' number of bytes into the pipe to process 'which', from the memory location 'mesg'. Since the message length is a variable, four bytes containing the message length are prepended to the message when it is written in the pipe. In case of successful writing operation, the routine returns a value 0 or else it returns -1 as an error condition.

READ_FROM subroutine

```
int read_from (which, mesg, len);
int which; char *mesg;
int *len;
```

This routine causes the process to read the message length indicated by memory location 'len' and then reads that many number of bytes into the memory location mesg, from the pipe specified by 'which'. As this is a blocked read instruction, execution is suspended till the number of bytes indicated by memory location len are read from the pipe. If all the requested bytes are read, read_from routine returns 0 indicating the success of read operation or else it returns -1.

Now let us discuss the routines which are called by the processes for interprocessor communication in case of broadcast communication.

```
BREAD(channel, mesg, len)

int channel;
char *mesg;
int *len;
```

This routine causes, number of bytes indicated by memory location 'len' to be read from the file, 'channel' into the memory location 'mesg'. In order to overcome the problem of inconsistency, the process, opens the shared file 'channel', and locks it and then reads the message and then unlocks it, releasing the file for other processors connected on the same bus. If the reading operation is successful then the routine returns 0 or else it returns -1.

```
BWRITE(channel, mesg, len)

int channel;
char *mesg;
int len;
```

This routine causes, 'len' number of bytes to be written to the file, 'channel' from the memory location 'mesg'. The process, opens the shared file 'channel', and locks it and then writes the message and then unlocks it, releasing the file for other processes connected on the same bus. If the writing operation is successful then the routine returns 0 or else it returns -1.

3.2.3 Support Routines

In this section, we describe number of support routines. These routines simplify the task of writing structure core to any topology of muticomputer network. We start with the routine connected() which is used to verify the connectivity of two given processing elements.

CONNECTED subroutine

```
#define TRUE 1
#define false 0
int connected(a,b)
int a,b;
```

The subroutine returns TRUE if the processing elements whose node ids are 'a' and 'b', are connected by a link, otherwise a FALSE value is returned.

CONNECT subroutine

```
connect(a,b)
int a,b;
```

The subroutine establishes an unidirectional link between processing elements whose nodeids are 'a' and 'b'.

BROAD_LINK subroutine

```
broad_link(a,b);
int a,b;
```

This subroutine establishes,a broadcasting link between the processing elements whose nodeids are 'a' and 'b'.

COMPLIMENT subroutine

```
int compliment(nodeaddr,i,d);
int nodeaddr,i,d;
```

This routine inverts the ith digit of node identifier nodeaddr, having 'd' number of digits , and returns the inverted node address.

GET_DIGIT subroutine

```
int get-digit(nodeaddr,r,d,i);
int nodeaddr,r,d,i;
```

Subroutine get_digit returns the ith digit from radix 'r' representation of the node identifier nodeaddr, having 'd' digits.

```
REPLACE(nodeaddr,r,i,j)  subroutine
  int nodeaddr,r,i,j;
```

Subroutine replace substitutes ith digit of the node identifier 'nodeaddr' by the digit 'j' and returns substituted node identifier nodeaddr.

3.3 STRUCTURE CORE

Having described the basic routines, to simulate single processing element of multicomputers we now describe the routines in the second layer of simulator. The second layer is built on top of the basic routines of simulator core and provides better user interface.

The structure core contains the following modules:

a. Input interface

b. Topology setup

c. Process Creation

d. Interprocessor Communication

e. Support routines

3.3.1 Input interface

The macro input() asks for the parameter and reads the parameter. Then the total number of processors in the network, is computed.

3.3.2 Topology Setup

The simulator core provides number of support routines to describe the network topology. Number of examples of the different multicomputer configurations are discussed in the next chapter. This network topology is passed to the simulator core to create the processes with appropriate communication links.

3.3.3 Process Creation

Different types of process creation calls are developed depending upon the topology of the multicomputers. All these routines have been implemented using the basic routine of process creation pfork().These are listed below:

subroutine SFORK

```
int sfork(dest);
int dest;
```

The sfork call is used in the networks like, ring or linear array of processing elements to create the processes sequentially. sfork returns 0 for normal operation and returns -1 for an error condition.

subroutine LFORK

```
int lfork(parameter);
int parameter;
```

The lfork call is used in multicomputer networks to create the process in accordance of the parameter. Here the parameter can be dimension(hypercube,MMS), level(tree), direction(mesh) etc. The call creates all the processes in the specified parameter. The routine returns 0 in normal execution and -1 for an error condition.

subroutine GFORK

```
int gfork();
```

The gfork call is used to create all the processes of the multicomputer network. This routine is used to create the processes in the networks of broadcast communication. On successful execution, this routine returns 0 or else it returns -1 as an error condition.

3.3.4 Interprocessor Communication

In this class of routines, data communication among processing elements is handled by the enhanced set of interprocess communication routines. These routines are developed using the basic routines of communication, write_to and read_from in case of point to point communication, bread and bwrite in case of broadcast communication.

Point to point communication:

subroutine LSEND_TO

```
int lsend_to(src,param,dataptr,len);
int src,param;
char *dataptr;
int *len;
```

This routine causes processing element 'src' to send 'len' number of bytes to the processing element connected in parameter 'param'. This parameter can be level(tree), dimension(hypercube,MMS), direction(mesh) etc. The complimentary routine, to receive the data is handled by lrecv_from wherein 'len' number of bytes are read by the processing element 'dest'.

subroutine LRECV_FROM

```
int lsend_to(dest,param,dataptr,len);
int dest,param;
char dataptr;
int len;
```

These routines have been implemented in accordance with the parameter in the interface to different topologies in the next chapter EXAMPLES.

Broadcast communication:

subroutine GSEND_TO

```
int gsend_to(src,dim,dataptr,len);
int src,dim;
char*dataptr;
int len;
```

subroutine GRECV_FROM

```
int grecv_from(dest,dim,dataptr,len);
int dest,dim;
char *dataptr;
int *len;
```

The gsend_to routine is used by the processing element 'src' to send data to all processing elements connected to the broadcast bus in dimension 'dim'. Whereas the complimentary routine grecv_from causes the processing element 'dest' to read the data from the broadcast bus connected to it in dimension 'dim'. These routines can be implemented by using the the basic routines of broadcast communication bread and bwrite. The implementation is illustrated in the next chapter EXAMPLES under structure interface to broad–cast hypercube.

3.4 CONCLUSION

In this chapter we presented the implementation of simulator suitable for all multicomputer configurations. The implementation is divided into two layers. The first layer is an implementation of process creation and interprocessor communication to simulate single processing element.

The second layer which is built over the first provides better user interface. The subroutine pfork, terminate, clean, connected,read_from, write_to implement general purpose routines and other subroutines described in this chapter are specific to the particular class of multicomputers. The simulator has been implemented on a sun 3/60 work station running sun operating system version 4.0.3c.

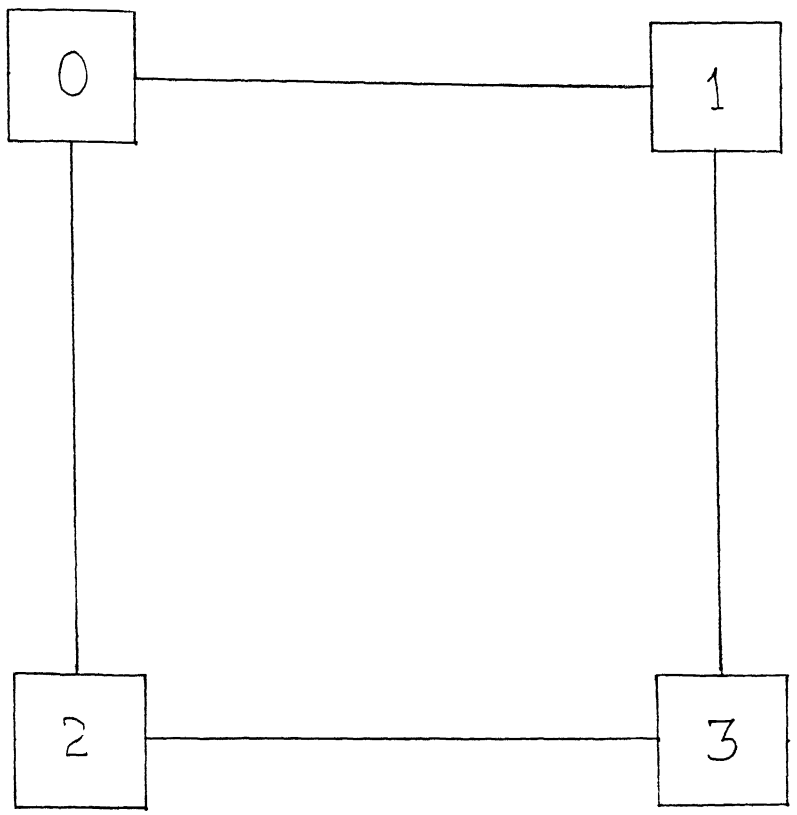

fig 3.1 Hypercube of dim = 2.

```
procedure open_pipes_1   /* Executed by each process */

 begin
 for all processors from 0 to N-1  do
  begin
    if  connected  then
     begin
       open pipe from this processor for reading
       open pipe to this processor for writing
     end
  end
end
```

Figure 3.2 : A naive algorithm for opening pipes.

```
procedure  open_pipes_by_proc  i   //  Executed  by  each
                                       process //

  begin
    for all processors from j = 0 to N-1  do
     begin
      if  connected  then
        if (i > j)
           begin
             open pipe from this processor for reading
             open  pipe  to  this  processor for writing
           end
        else
          begin
             open pipe to this processor for writing
             open pipe from this processor for reading
          end
       end
    end
```

Figure 3.3 : Dead_lock avoiding algorithm for opening pipes.

CHAPTER 4: EXAMPLES

4.1 INTRODUCTION

The major objective of the thesis is to provide facilities for simulation of multicomputers. A multicomputer is characterized by number of processing elements and a set of data routing functions provided by the interconnection network. The processing elements are independent of the network topology and therefore are implemented by the simulator core. The interconnection network is specific to a class of multicomputers and is described by the structure core.

The interconnection functions are different for different machines. So in order to provide a general purpose simulator tool, we maintain structure interface for each topology. In this chapter we discuss different examples of such interfaces. In section 4.2 we discuss the hypercube structure core, MMS (Multidimensional Multilink System) structure core in section 4.3, mesh[Hwang85] structure core in section 4.4,binary tree structure core in section 4.5 and lastly in section 4.6 the broadcast hypercube structure core . Finally we conclude in section 4.6.

The structure core consists of the following modules:

i) Input interface

ii) Topology setup

iii) Process creation

iv) Interprocess communication

v) Support routines

Let us discuss these modules with respect to different topologies.

4.2 STRUCTURE INTERFACE TO HYPERCUBE[hp_cube.c]

The network consists of $N = 2^d$ nodes forming a 'd' dimensional hypercube. The nodes are labeled $0, 1, \ldots 2^d - 1$. Two nodes are adjacent if their labels differ in exactly one bit position. Fig 4.1 shows a hypercube model.

(i) Input interface: The macro, input(d) reads dimension 'd' as an input. Then the total number of processors present in the network , $N = 2^d$.

(ii) Topology set_up: Number of support routines are provided in the simulator core to facilitate the set up of topology of different multicomputer configurations. The hypercube network has been established using these routines as as follows :

```
for peid = 0 to N-1
   for j = 0 to d-1
     connect(peid,compliment(peid,j));
```

Where peid is the node address of the processing element.

(iii) Process creation: As described in the previous chapter, in section structure core, the routine lfork is used to create the process dimension wise.

```
LFORK subroutine
lfork (dim);
in dim;
```

The lfork call creates all the processes in dimension 'dim'. The subroutine is implemented by the basic routine of

(iv) Interprocess communication: Using the basic routines of communication, read_from, write_to implemented in simulator core, the subroutine lsend_to is developed to provide communication for any process to its neighbor in the dimension specified by 'dim'. These routines return 0 on successful execution else return-1 as an error condition.

LSEND_TO subroutine

```
int lsend_to (src, dim, dataptr, len);
int src, dim;
char *dataptr;
int len;
```

LRECV_FROM subroutine

```
int lrecv_from (dest, dim, dataptr, len);
int dest, dim;
char *dataptr;
int len;
```

The subroutine lsend_to allows 'len' number of bytes to be sent by the processing element whose address is specified by 'src' to the processor connected to 'src' in dimension 'dim'. The complimentary routine for receiving data has been implemented by the routine lrecv_from(). These routines are implemented using the basic routines of interprocessor communication, as follows:

```
int lsend_to (src, dim, dataptr, len);

int src, dim;
char *dataptr;
int len;

{
  int neigb;

  neigb = compliment (src,dim);
  write_to (neigb, dataptr, len);
}
```

```
lrecv_from (dest, dim, dataptr, len)

int dest, dim;
char *dataptr;
int len;

{
  int neigb;

  neigb = compliment (dest,dim);
  read_from(neigb, dataptr, dim);
}
```

(v) Support routines: Number of subroutines in this section include calls for obtaining topological parameters to simplify the task of user programming. We start with the get_nodeid subroutine used to get the node address of the processing element.

GET_NODEID subroutine

```
int get_node_id();
```

This routine returns the node address of the process being executed.

GET_DIM subroutine

```
int get_dim();
```

The subroutine returns the dimension of the hypercube network.

GET_NOPROCS subroutine

```
int get_noprocs();
```

The routine get_noprocs returns the number processors in the hypercube network.

4.3 STRUCTURE INTERFACE TO MMS[mms.c]

The model consists of $N = p^d$ nodes forming a MMS network in dimension 'd' of drop 'p'. The processors are

connected if only if the addresses of the processors are differed by single bit. Fig 4.2 illustrates this model. The processors are numbered from 0 to N-1.

(i) Input interface: The macro, input(d) reads the dimension and the macro input(p) reads the drop of the network. Then the total number of processors in the network = p^d.

(ii) Topology set_up: The topology of the MMS network is set up by using the support routines provided in the simulator as follows:

```
for peid = 0 to N-1
   for j= 0 to d-1
    for i = 0 to p-1
       if (get_digit(peid,j,p) != i)
       connect(peid,replace(peid,p,j,i))
```

Where peid is the node address of the processing element. This topology of the network is passed to the simulator core to create the processes with the appropriate communication links.

(iii) Process creation: The routine lfork is implemented to create the processes dimensionwise. It is same as in the case of hypercube network, since, hypercube is the special case of MMS structure , wherein drop of the network is always two.

(iv) Interprocess communication: The subroutine lsend_to and lrecv_from are developed to provide communication between the process and its neighbors in the dimension specified by the argument 'dim' and drop 'dr'. These routines return 0 indicating the success of the operation or else it return -1.

LSEND_TO subroutine

```
int lsend_to (src, dim, dr, dataptr, len);
int src, dim, dr;
char *dataptr;
int len;
```

LRECV_FROM subroutine

```
int lrecv_from (dest, dim, dr, dataptr, len);
int dest, dim, dr;
char *dataptr;
int len;
```

These routines are implemented by using the basic subroutines of interprocess communication write_to and read_from as follows.

```
lsend_to(src,dim,dr,dataptr,len)

int src,dim,dr;
char *dataptr;
int len;

 {
   int neighb;
   neighb = replace(src,p,dim,dr);
   write_to(neighb,dataptr,len);
 }


lrecv_from(dest,dim,dr,dataptr,len)

int dest,dim,dr;
char *dataptr;
int len;

 {
   int neighb;
   neighb = replace(dest,p,dim,dr);
   read_from(neighb,dataptr,len);
 }
```

(v) Support routines: The subroutine get_drop is implemented here in addition to the routines that are explained in hypercube structure, to get the drop of the MMS network in the user program.

GET_DROP() subroutine

```
int get_drop();
```

4.4 STRUCTURE INTERFACE TO MESH[mesh.c]

In a mesh network, The nodes are arranged into a two dimensional lattice. Communication is allowed only between neighboring nodes; hence interior nodes communicate with four other processors. Fig 4.3 illustrates a 2_d mesh network with no wraparound connections. Let 'n' be the size of the mesh. Let N be the total number of processing elements in the network. The processors are numbered from 0 to N-1.

(i) Input interface: The macro input(size) reads the size of the mesh to be described. Then total number of processors in the network, N = n*n.

(ii) Topology setup: The network of the 2-d mesh using the support routines of the simulator core can be established as follows:

```
for peid = 0 to N-1
  connect(peid,get_neighb(peid,LEFT));
  connect(peid,get_neighb(peid,RIGHT));
  connect(peid,get_neighb(peid,ABOVE));
  connect(peid,get_neighb(peid,BELOW));
```

The routine get_neighb is described later on under support routines. Once structure of the network is established, it is passed to simulator core, to create processes and the appropriate communication links.

(iii) Process creation: The subroutine lfork() is implemented to create the processes row wise.

```
lfork subroutine

int lfork(row);
int row;
```

This routine has been implemented using the basic routine of process creation pfork().

(iv) Interprocessor communication: The complimentary subroutines for communication between each processor of the network, to its neighbor in the different directions like, LEFT, RIGHT, BELOW, ABOVE, lsend_to and lrecv_from have been developed. On successful completion of the data transfer, the routines return 0 or else they return -1 as an error condition.

```
subroutine LSEND_TO

int lsend_to(src,dir,dataptr,len);
int src, dir;
char *dataptr;
int len;


subroutine lrecv_from

int lrecv_from(dest,dir,dataptr,len);
int dest, dir;
char *dataptr;
int len;
```

These subroutines cause the processing element to send the data to or receive the data from the processor connected to it, in the direction 'dir'. An implementation of these routines using the basic routines of interprocessor communication is given below.

```
int lsend_to(src,dir,dataptr, len);
int src,dir;
char *dataptr;
int len;

{
 int neighb;
  neighb = get_neighb(src,dir);
  write_to(neighb,dataptr,len);
}
```

```
int lrecv_from(dest,dir,dataptr, len);
int dest,dir;
char *dataptr;
int len;

{
 int neighb;

  neighb = get_neighb(dest,dir);
  read_from(neighb,dataptr,len);
}
```

The get_neighb() is described later on in this section under support routines.

(v) Support routines : The routines to get the parameters of the 2_d mesh network are implemented here. The routine get_size() returns the size(no. of rows or columns) of the 2_d mesh network. The routine get_noprocs() and get_nodeid() are same as in the case of hypercube structure core.

Subroutine get_neighb is developed to get the neighbor of the processor connected to it, in direction dir.

Subroutine GET_NEIGHB

```
int get_neighb(nodeaddr,dir)
int nodeaddr,dir;
```

This routine checks whether the processor with node address nodeaddr has neighbor in the direction 'dir', if so it then returns node address of the processor connected to it in direction 'dir'(LEFT,RIGHT,ABOVE,BELOW) or else it returns -1;

4.5 STRUCTURE INTERFACE TO BINARY TREE[tree.c]

The network consisting of $N = 2^{l} - 1$ nodes forms a binary tree of height l. Communication is allowed only between

their parent and children. Fig 4.4 shows a binary tree. The processors are numbered from 0 to N-1.

i) Input interface: The macro input(l) reads the no. of levels(height) of the binary tree network. Then total number of processors in the network, $N = 2^{l} - 1$.

ii) Topology setup: The topology of the network is established by using the support routines of simulator core as follows:

```
for peid = 0 to N - 1

    get_parent(peid)
    connect(peid,get_parent(peid))
    connect(peid,get_left_child(peid))
    connect(peid,get_right_child(peid))
```

The routines get_parent, get_left_child and get_right_child are described later under support routines. Once the network is established, it is passed to the simulator to create the processes with communication links accordingly.

iii) Process creation : The routine lfork creates the processes level by level. This routine has been implemented by the basic routine of process creation pfork. It returns 0 on successful creation of the processes or else it returns -1 as an error condition.

```
subroutine LFORK

int lfork(level);
int level;
```

iv) Interprocess communication: Number of interprocess

communication for the processor with its parent and its children of the binary tree network.

Subroutine SEND_TO_PARENT

```
int send_to_parent(src,dataptr,len);
int src; char *dataptr; int len;
```

Subroutine RECV_FROM_PARENT

```
int recv_from_parent(dest,dataptr,len);
int dest; char *dataptr; int len;
```

The subroutine send_to_parent causes the data to be sent from the processor specified by the node address 'src' to its parent. The complimentary routine recv_from_parent makes the processor 'dest', to receive the data from its parent. These routines, return 0 on successful execution or else they return -1 as an error condition. These routines are implemented by using the basic routines of communication read_from,write_to as follows:

```
int send_to_parent(src,dataptr,len)
int src;
char *dataptr;
int len;

{
  int dest;

  dest = get_parent(src);
  write_to(dest,dataptr,len);
}

int recv_from_parent(dest,dataptr,len)
int dest;
char *dataptr;
int len;

{
  int src;

  src = get_parent(dest);
  read_from(src,dataptr,len);
}
```

Similarly the routines $end_to_left_child, recv_from_left_child, send_to_ right_child and recv_from_right_child are implemented to provide communication between the processor and its child. Please refer tree.c in APPENDIX B for details.

v) Support routines: The subroutine get_height(),called by the user program ,is implemented to get the number of levels in the binary tree network.

Subroutine GET_HEIGHT

```
int get_height();
```

The subroutines get_noprocs(), get_nodeid() are same as that of other structure cores. In addition to these, the subroutines get_parent,get_left_child, get_right_child are implemented to ease the writing of topology setup and to develop the routines of interprocess communication.

Subroutine GET_PARENT

```
int get_parent(nodeaddr)
int nodeadddr;
```

get_parent checks, whether the processor specified by node address, 'nodeaddr' is a root of the tree . if so, it returns -1 or else it returns the node address of its parent processor.

Subroutine GET_LEFT_CHILD

```
int get_left_child(nodeaddr);
int nodeaddr;
```

get_left_child verifies. whether the processor specified by node address, 'nodeaddr' is a leaf node. If so, it returns -1 or else it returns the node address of its left

Subroutine GET_RIGHT_CHILD

```
int get_right_child(nodeaddr);
int nodeaddr;
```

get_right_child verifies, whether the processor specified by node address, nodeaddr is a leaf node. If so, it returns -1 or else it returns the node address of its right child processor.

Now let us go for the structure core for broad_cast communication network. We discuss only example of it i.e. structure interface to broad_cast hypercube.

4.6 STRUCTURE INTERFACE TO BROADCAST HYPERCUBE [br_hpcube.c]

Fig 4.5 shows a broad_cast hypercube model . Let 'd' be the dimension, let 'p' be the drop of the of the hypercube network . Let 'N' be the total number of processors in the network. The processors are numbered from 0 to N-1.

i) Input interface: It is same as in the case of interface to MMS structure with point to point communication.

ii) Topology setup: Using the support routines of simulator core the network of broadcast hypercube is established as follows:

```
for peid = 0 to N-1
 for j = 0 to d -1
  for i = 0 to p-1
   if(get_digit(peid,j,p) != i)
     broad_link(peid,replace(peid,p,j,i),get_channel(peid,j));
```

The routine get_channel is described later on under support routines. The topology of the network is passed to the simulator to create the processes with broadcast links in

iii) Process creation: The gfork routine creates the processes dimension wise creating files, on for each broadcast channel. This routine has been implemented using the basic routine of process creation pfork. The routine returns 0 on successful creation of processes or else it returns -1 as an error condition.

Subroutine GFORK

```
int gfork(dim);
int dim;
```

iv) Interprocess communication: The routine gsend_to causes the processor 'src' to send the data on the broadcast bus connected to it in dimension 'dim'. The complimentary routine grecv_from makes processor dest to receive the data from the broadcast bus connected to it in dimension 'dim'. If the operation of data transfer is successful the routines return 0 or else they return -1 as an error condition.

Subroutine GSEND_TO

```
int gsend_to(src,dim,dataptr,len);
int dim,src; char dataptr; int len;
```

Subroutine GRECV_FROM

```
int grecv_from(dest,dim,dataptr,len);
int dim,dest; char dataptr; int len;
```

These routines are implemented by using the basic routines of broadcast communication bread and bwrite as follows:

```
int gsend_to(src,dim,dataptr,len)

int src,dim;
char dataptr;
int len;
```

```
{
  int ch;

   ch = get_channel(src,dim);
   bwrite(src,ch,dataptr,len);
}
```

```
int grecv_from(dest,dim,dataptr,len)

 int dest,dim;
 char dataptr;
 int len;

   {
     int ch;

     ch = get_channel(src,dim);
     bread(dest,ch,dataptr,len);

   }
```

v) Support routines: The routines get_drop, get_dim, get_noprocs are the same as in the case of interface to hypercube with point to point communication.

In addition to these, a support routine get_channel has been implemented.

Subroutine GET_CHANNEL

```
int get_channel(nodeaddr,dim);
int nodeaddr,dim;
```

This routine returns the channel number to which the processor, with node address 'nodeaddr' is connected in dimension 'dim'.

4.7 CONCLUSION

This chapter has discussed four multicomputer configurations of point to point communication, namely hypercube, MMS, 2_d mesh, binary tree and one multicomputer network of broadcast communication i.e. broadcast hypercube model. To provide better user interface, each structure file consists of an input interface to read the parameters of the network, and interfaces for process creation and interprocess communication. Number of support routines are available to get the parameters of the network in the user program.

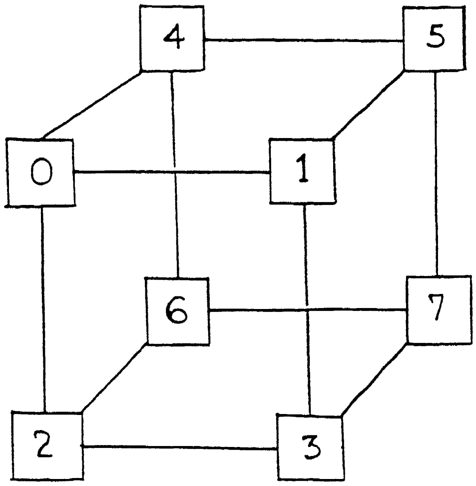

fig 4.1 Hypercube structure of dim=3

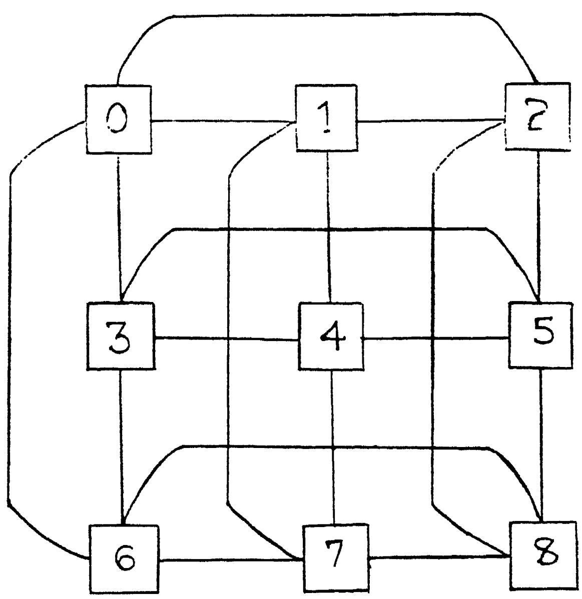

fig. 4.2. MMS structure of drop=3, dim=2.

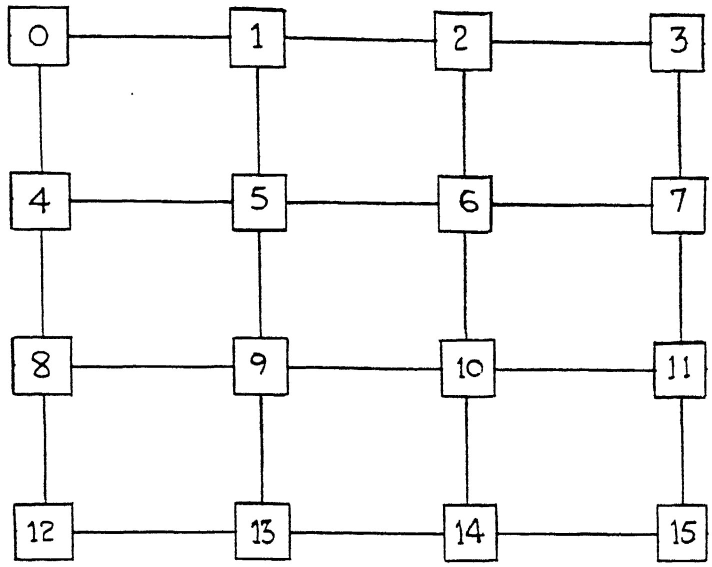

fig 4·3 2-d Mesh of size = 3.

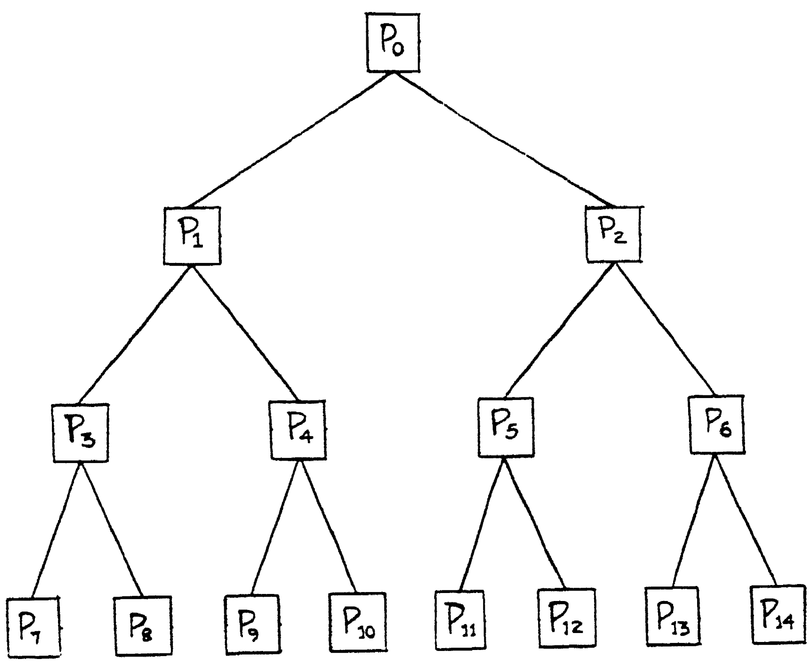

fig 4·4 Binary Tree of height = 4.

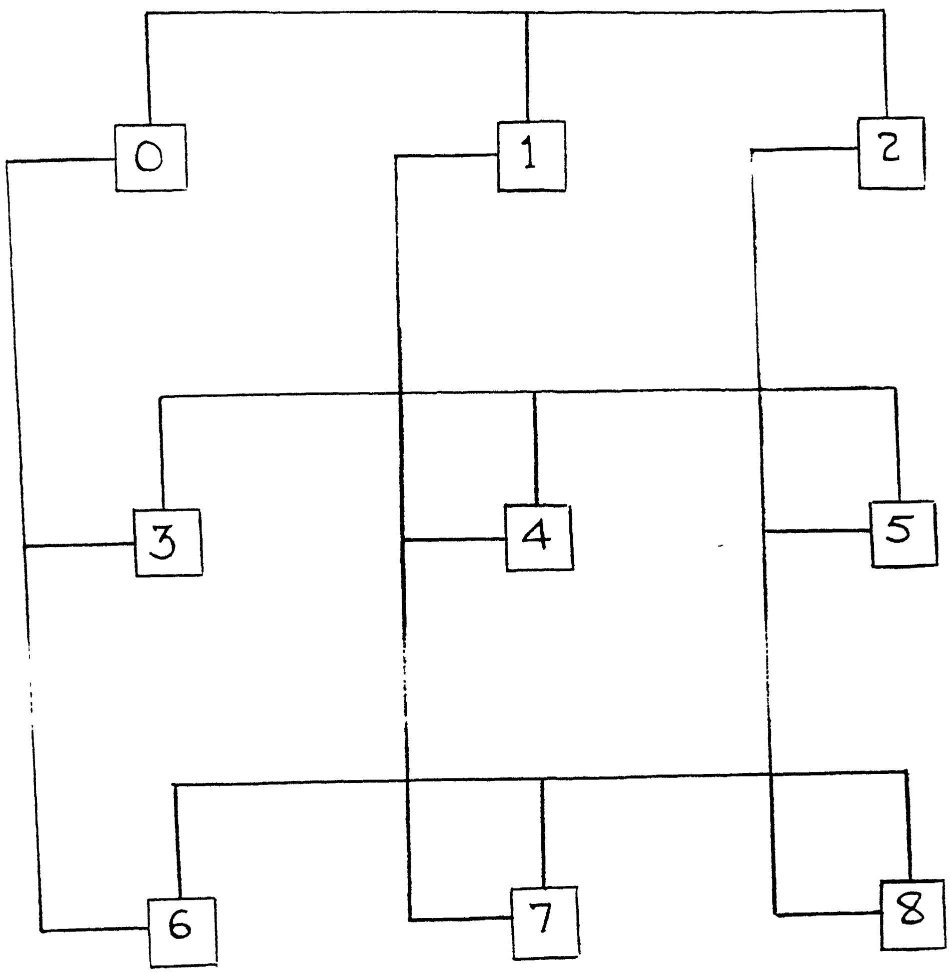

fig 4.5. Broadcast Hypercube of dim = 2.

CHAPTER 5 : SIMULATION RUNS

5.1 INTRODUCTION

In this chapter, we discuss the sample programs of few parallel algorithms, that are implemented using the simulator package. In section 5.2, we discuss the implementation of summation of numbers on hypercube architecture, summation of numbers on mesh structure in section 5.3 and matrix_by_vector multiplication on tree structure in section 5.4. Finally we conclude in section 5.4.

5.2 SUMMATION OF NUMBERS ON HYPERCUBE[Quinn87]

The algorithm to add $n = 2^m$ values on a hypercube model has been adapted from [Quinn87] .

```
procedure SUMMATION(n)
/* computes a_0 + a_1 + a_2 + ... + a_n-1
   result is stored in a_0    */
begin
  for i = logn-1 downto 0 do  /* i : dimension number */
      d = 2^i
   for j = 0 to d-1 do in parallel
      t_j  <= a_j+d
      a_j  <- a_j +t_j
  endfor
  endfor
end
```

In the algorithm above, communication of the data item from an adjacent processor's local memory into the active

Since, every loop iteration requires constant time, the complexity of this algorithm is O(logn). The algorithm is illustrated in fig 5.2.1 for n =16.

This algorithm can be implemented using the primitives available in the concerned structure file [hp_cube.c]. The routine lfork(dim) is called to invoke the simulator to create the processes dimension wise and the routines lsend_to and lrecv_from by the process to communicate to its neighbour in the given dimension. The user's program implementing the above algorithm is illustrated in Fig 5.2.2.

5.3 SUMMATION OF NUMBERS ON 2_d MESH STRUCTURE

An algorithm[Quinn87] to do the same task on a 2_d mesh connected model is given below. $n=l^2$, where l be the number of rows (or columns) in the model. For simplicity the n values to be added are stored, one per processing element. The algorithm works by summing all the rows in column 1 and then summing column 1.

When the algorithm concludes the element $a_{1,1}$ contains the sum.

ADDITION (2_d mesh)

begin

for i <- l-1 down to 0 do

 for all $P_{,1}$ where $1 \le j \le l$ do

  $t_{j,i}$ <= $a_{j,i+1}$ /* column i active*/

  $a_{j,i}$ <- $a_{j,i}$ + $t_{j,i}$

 endfor

endfor

```
for i <- l-1 downto 0 do
  for all P_{i,l}   do   /*only a single processing element
   t_{i,l} <= a_{i+1,l}               in column l is active */
   a_{i,l} <- a_{i,l} + t_{i,l}
  endfor
endfor
end
```

This algorithm has been successfully implemented and tested using the simulator package. The routine lfork(row) is called for process creation row wise and lsend_to and lrecv_from for interprocess communication. These routines are available in the corresponding structure file [mesh.c]. The program implementing the summation algorithm is illustrated in fig 5.2.2.

5.4 MATRIX_BY_VECTOR MULTIPLICATION ON TREE STRUCTURE

The problem addressed in this section is that of multiplying an m X n matrix A by an n X 1 vector U to produce an m X 1 vector V. Matrix_by_vector multiplication requires m+n-1 steps on a linear array. It is possible to reduce this time to m - 1+logn by performing the multiplication on a tree connected network.

The algorithm[Akl89] is given as a procedure TREE MV MULTIPLICATION.

```
procedure TREE MV MULTIPLICATION(A,U,V)

do steps in parallel
  (1) for i = 1 to n do in parallel
      for j = 1 to m do in parallel
      (1.1) compute u_i * a_{j,i}
      (1.2) send results to parent
      endfor
```

```
(2) for i = n+1 to 2n-1 do in parallel
    while P_i receives two inputs do
     (2.1) compute the sum of the two inputs
     (2.2) if i < 2n -1 then send the result to parent
                else produce the result as output.
                endif
     endwhile
   endfor
```

This algorithm is illustrated in fig 5.4.1 for n=3.

This algorithm can be easily implemented calling the routines available in tree structure file [tree.c], lfork(level) for creating the processes level wise, lsend_to_parent and lrecv_from_rhild and lrecv_from_rchild for interprocess communication. The user program implementing the above algorithm is illustrated in fig 5.4.2.

5.5 CONCLUSION

The simulator, can be easily used as a test bed to verify the parallel algorithms, for different multicomputer architectures. User should use only the routines that are available in the concerned predefined structure file. To run the user program he should read the instructions given in the appendix C.

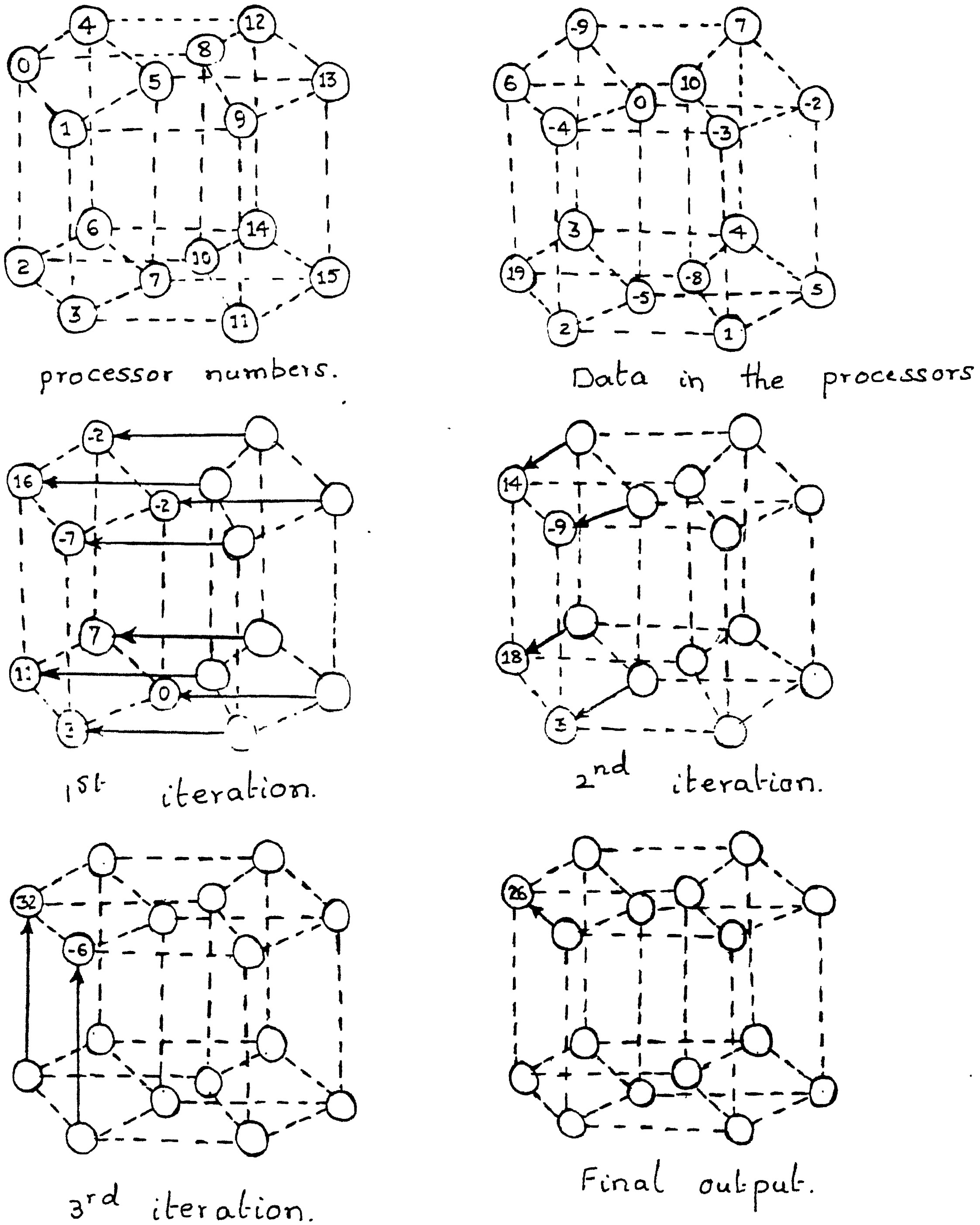

fig 5.2.1 Illustration of summation of numbers on hypercube.

```
SUMMATION(numbers)
int numbers[size];

{
   int lsum,dim,d,k,dl;
   int len,from_source,node_id;

    for( dl = 0; dl < get_dim(); dl++)
      lfork(dl);

    node_id = get_nodeid();

    dim = get_dim()-1;

                               /*add the numbers dimension_vise*/
    lsum = numbers[node_id];
    for( d = dim; d >= 0; d--)
      {
        k = power(2,d);
        if ( node_id >= k)
          {
            lsend_to(node_id,d,&lsum,sizeof(int));
            terminate();
          }
        else
          {
            lrecv_from(node_id,d,&from_source,&len);
            lsum += from_source ;
          }
       }
   if( node_id == 0)
   return lsum;
 }
```

Figure 5.2.2 : User program for summation on hypercube

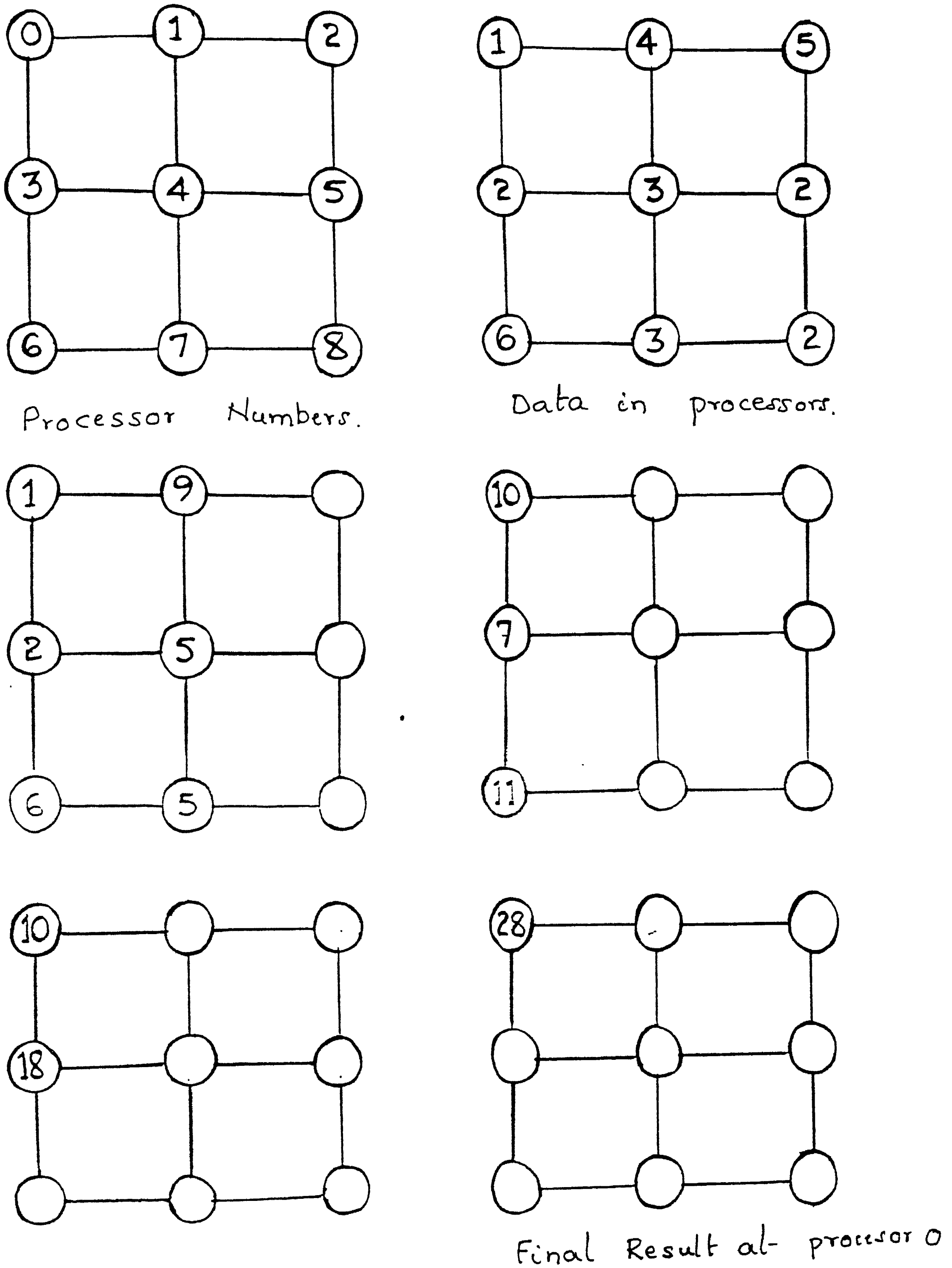

fig. 5.3.2. Illustration of summation of numbers on 2-d Mesh.

```
MESH_SUMMATION (numbers)
int numbers[size];

{
 int lsum,len,node_id,from_source;
 int r,kl,row,sizel,column;

 for( r = 0; r < sizel; r++)        /*create the process
   lfork(r);                                  row wise */
 node_id = get_nodeid();
 sizel = get_meshsize();
lsum = numbers[node_id];
                        /*add the numbers column_vise*/

 for( column = sizel-1; column > 0; column--)
 {
  if ( node_id % sizel == column)
   {
    lsend_to(node_id,LEFT,&lsum,sizeof(int));
    terminate();
   }

  else
    if((node_id % sizel) == column-1)
     {
      lrecv_from(node_id,RIGHT,&from_source,&len);
      lsum += from_source ;
     }
 }

                        /* only first column is active */
 for( row = sizel-1; row > 0; row--)
 {
  if((node_id /sizel == row) &&(node_id % sizel == 0))
   {
    lsend_to(node_id,ABOVE,&lsum,sizeof(int));
    terminate();
   }


  else

    if((node_id/sizel == row-1)&&(node_id % sizel == 0))

     {
      kl = node_id +sizel;
      lrecv_from(node_id,BELOW,&from_source,&len);
      lsum += from_source ;
     }
 }

 if( node_id == 0)
   return lsum;
 }
```

Figure 5.3.2: User program for summation
on 2_d mesh

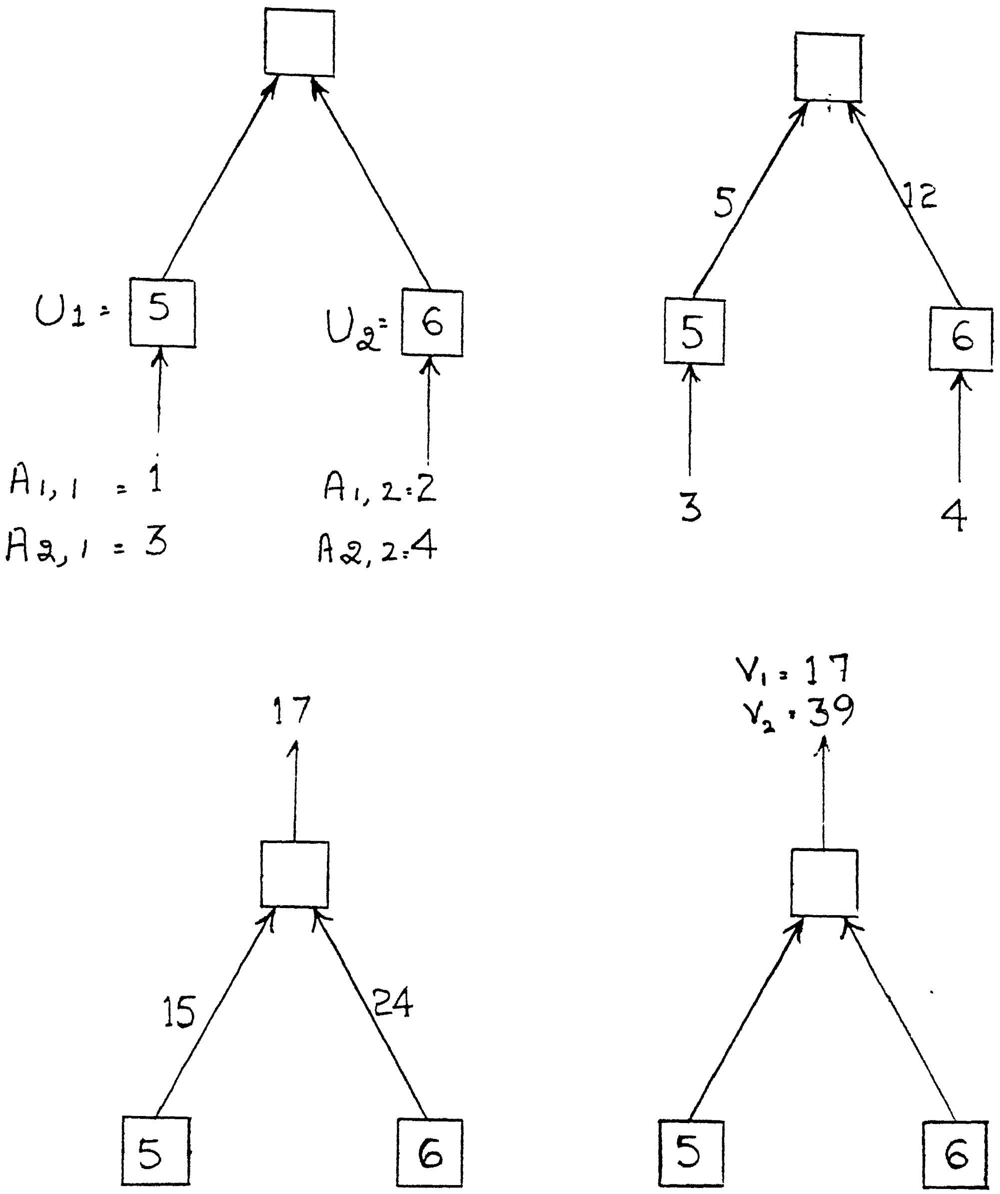

fig. 5.4.1 Illustration of matrix_by_Vector multiplication on tree.

```
matrix_by_vector(A,U,V)
int A[size][size1];
int U[size1];
{
 int l,i,n, m = 0;
 int node_id,len;
 int lsource,rsource;

   for( l = 1; l < get_height(); l++)
   lfork(l);
   node_id = get_nodeid();
   l =  get_height();
   for( i = 0 ; i < n; i++)
    {
                 /* Compute results and send to parent */
     if(leaf_node(node_id))
      {
       index = node_id - (power(2,l-1)-1) ;
       product = U[index]  * A[i][index];
       send_to_parent(node_id, &product,sizeof(int));
      }

     else
      {
                 /*Receives two inputs and find the sum*/
       recv_from_leftchild(node_id,&lsource,&len);
       recv_from_rightchild(node_id,&rsource,&len);
       product1 = lsource + rsource;
       if(node_id != 0)        /* If not root node send the
                                     result to parent */
        send_to_parent(node_id, &product1,sizeof(int));

       else
        {
         V[m]  =   product1; /*produce  the  result  as
         m++;                                 output*/
        } /*else*/
       } /*else*/

     } /*for*/
 }
```

Figure 5.4.2 : Matrix_by_vector multiplication on tree structure

CHAPTER 6: CONCLUSIONS

In this thesis, a simulator package is developed. It is a platform for testing users parallel algorithms written to run on multicomputer architectures. In order to make this package complete on its own, some features are to be added. A debugger to correct user's program is dealt in section 6.1, a performance monitor is discussed in section 6.2. Finally the shortcoming of this simulator is discussed in section 6.3.

6.1 DEBUGGER

In this simulator, we have used signals to take care of errors occurred during simulation. Wherein, the process at which error has occurred sends the signal to the parent process (process 0) which in turn distributes the signal among all processes created, terminating them with error condition. Added to these we should provide a debugger for the user in such a way that he should be able to correct his program with least difficulty.

6.2 PERFORMANCE MONITOR

In most of the cases, a specific multicomputer network is more feasible than the rest for the given parallel algorithm. Hence a facility can be provided, so that the simulator can evaluate the performance of the user's program on different multicomputer networks and should come out with the most efficient one with minimum communication costs.

6.3 SHORTCOMINGS

The main shortcoming of the implementation is that, the size of multicomputer network that can be simulated is limited. This is because, the operating system imposes an upper limit on the total number of processes and also on the number of processes that a single user can have running at the same time[Sun88]. Thus simulator can't simulate very large network (bigger than 64 processors). This shortcoming can be removed with the usage of user level thread package.

APPENDIX A

Routines available to write structure file

NAME

connect - establish the link between two processors.

SYNOPSIS

```
#include "structure.h"
void connect(a,b)
int a,b;
```

DESCRIPTION

Connect establishes the unidirectional link between two processing elements whose node ids are 'a' and 'b'. This subroutine is used to set up the topology of point to point connected multicomputer network.

RETURN VALUE

None.

NAME

connected - checks the connectivity between processing elements.

SYNOPSIS

```
#include "structure.h"
int connected(a,b)
int a,b;
```

DESCRIPTION

Connected returns TRUE if the processing elements whose node ids are 'a' and 'b' are connected else returns FALSE if not connected.

RETURN VALUE

Returns 1 if connected else returns 0.

NAME

broad_link - establishes broadcast link between the two processing elements.

SYNOPSIS

```
#include "structure.h"
 void broad_link(a,b,ch)
 int a,b,ch;
```

DESCRIPTION

Broadcast link establishes broadcast link, 'ch' between processing elements whose node ids are 'a' and 'b'. This routine is used to establish the structure of the broadcast communication network of multicomputers.

RETURN VALUE

None.

NAME

get_digit - extracts a specified digit from the address of nodeid.

SYNOPSIS

```
#include "structure.h"
int get_digit(nodeaddr,r,d,i)
int nodeaddr,r,d,i;
```

DESCRIPTION

get_digit extracts the ith digit from the radix 'r' representation with 'd' digits in the node identifier nodeaddr.

RETURN VALUE

Returns the extracted digit.

NAME

compliment - inverts the specified digit of node address.

SYNOPSIS

```
int compliment(nodeaddr,d,i)
int nodeaddr,i,d;
```

DESCRIPTION

compliment inverts the ith digit of node address having 'd' digits, nodeaddr.

RETURN VALUE

Returns the complimented node address.

NAME

replace - substitutes the specified digit of node address by given digit.

SYNOPSIS

```
#include "structure.h"
int replace(nodeaddr,r,d,i,j)
int nodeaddr,r,d,i,j;
```

DESCRIPTION

replace, substitutes the ith digit of radix 'r' representation of the node identifier, having 'd' digits, nodeaddr by the digit 'j'.

RETURN VALUE

Returns the substituted node identifier, nodeaddr.

APPENDIX B: EXAMPLES OF STRUCTURE FILE

'hp_cube.c'

```
#include "structure.h"

 int dim;
 int proc_no;

 structmain()
  {
   int peid,id,neighb;

   input(dim)                  /*input interface*/
   no_procs = power(2,dim);
   initialize();
                               /* Topology Setup*/
   for(peid = 0; peid < no_procs; peid++)
     {
      for(id = 0; id < dim; id++)
        {
          neighb = compliment(peid,id,dim);
          connect(peid,neighb);
        }
     }

  start_sim();          /* Set up the simulation environment */
  main();              /* Entry point of user programme */
  clean();            /* Remove the communication links */
  terminate();       /* Terminate the process 0 */
  }

                  /*Process creation*/
 lfork(diml)
 int diml;

 {
   int np,k,p,pl;
    np = power(2,diml);

    if (get_nodeid() != 0)
     return;
     for(p = 0; p < np; p++) /* Create processes in
                                 dimension diml*/

       {
         k = p+np;
         pfork(k);
         pl = getpid();
         if(pl == child_pid)
          {
            proc_no = k;
            return ;
          }
       }
  }
```

```
                    /*Interprocess Communication*/
lsend_to(src,diml,mesg,len)

 int diml,src;
 char *mesg ;
 int len;

  {
   int i,dest;
   dest = compliment(src,diml,dim);
    write_to(dest,mesg,len);
  }




lrecv_from(dest,diml,mesg, len)

int dest,diml;
char *mesg ;
int *len;
  {
     int i,src;

    src = compliment(dest,diml,dim);
    read_from(src,mesg,len);
  }
                         /*Support Routines*/
get_dim()

  {
       return dim;
  }

get_noprocs()

  {
    return no_procs;
  }

get_nodeid()

 {
   return proc_no;
 }
```

'mms.c'

```
#include "structure.h"

int dim,drop;
int proc_no;

structmain()

 {
  int peid,d,p,neighb;
  char str[LENGTH];
                              /*input interface*/
  input(drop)
  input(dim)
  no_procs = power(drop,dim);
  initialize();
                              /*Topology setup*/
  for(peid = 0; peid < no_procs; peid++)
  {
   for(d =0; d < dim; d++)
    {
     for(p = 0; p < drop; p++)
      {
       if(get_digit(peid,drop,dim,d) != p)
         connect(peid, replace(peid,drop,dim,d,p));
      }
    }
  }

start_sim();          /* Set up the simulation environment */
main();              /* Entry point of the user programme */
clean();            /* Remove the communication links */
terminate();       /* Terminate the process 0 */

}

                    /*Process Creation*/
 lfork(diml)
 int diml;

 {

  int np,npl,k,p,pl;
  np = power(drop,diml) * (drop -1);
  npl = power(drop,diml);

  if( get_nodeid() != 0)
  return;
  for( p = 0; p < np ; p++)/*Create the processes
                            in dimension diml*/
   {
    k = p+npl;
    pfork(k);
    pl = getpid();
```

```
    if( pl == child_pid)
        {
          proc_no = k;
          return;
        }
    }
}
                              /*Interprocess Communication*/
lsend_to(src,diml,dr,mesg,len)

 int src,diml,dr;
 char *mesg;
 int len;

  {
   int i,dest;
   dest = replace(src,drop,dim,diml,dr);
   write_to(dest,mesg,len);
  }


 lrecv_from(dest,diml,dr,mesg,len)

 int dest,diml,dr;
 char *mesg;
 int *len;

 {
  int i,src;
  src = replace(dest,drop,dim,diml,dr);
  read_from(src,mesg,len);
 }                            /*Support Routines*/
  get_dim()
  {
   return dim;
  }

  get_drop()

   {
    return drop;
   }

   get_noprocs()

    {
     return no_procs;
    }

  get_nodeid()
   {
    return proc_no;
   }
```

'mesh.c'

```
#include "structure.h"

int mesh_size,proc_no;

 structmain()

 {

  int peid,neighb;

                              /*Input interface*/
  input(mesh_size)

  no_procs = mesh_size*mesh_size;

  initialize();

                             /* Topology Setup */
 for(peid = 0; peid < no_procs; peid++ )

  {
   if((neighb = get_neighb(peid,LEFT)) != -1)
   connect(peid, neighb);
   if((neighb = get_neighb(peid,RIGHT)) != -1)
   connect(peid, neighb);
   if((neighb = get_neighb(peid,ABOVE)) != -1)
   connect(peid, neighb);
   if((neighb = get_neighb(peid,BELOW))!= -1)
   connect(peid, neighb);
  }


start_sim();   /* Set up the simulation environment */
main();       /* Entry pont of the user programme */
clean();          /* Remove the communication links */
terminate();    /* Terminate the process 0 */

}



                                  /* Process Creation */
lfork(row)
int row;

{
 int np,npl,p,k,pl;

 if(row == 0)
  {
    npl = 1;
    np = mesh_size-1;
  }
 else
 {
   npl = row * mesh_size;
   np = mesh size;
```

```
 if(get_nodeid() != 0)
 return;

 for(p = 0; p < np; p++)          /*Create the processes
                                  in the given row */
   {
     k = p+npl;
     pfork(k);
     pl = getpid();
     if(pl == child_pid)
      {
        proc_no = k;
        return;
      }
   }
}


                       /* Interprocess Communication */
int lsend_to(src,dir,dataptr,len)
int src,dir;
char *dataptr;
int len;

{
  int dest;

  dest = get_neighb(src,dir);
  write_to(dest,dataptr,len);
}


int lrecv_from(dest,dir,dataptr,len)
int dest,dir;
char *dataptr;
int *len;

{
  int src;

  src = get_neighb(dest,dir);
  read_from(src,dataptr,len);
}


                       /* Support Routines */
get_nodeid()

{
 return proc_no;
}
get_noprocs()

{
return no_procs;

}
```

```
get_mesh_size()

{
  return mesh_size;
}


int get_neighb(peid,dir)
int peid,dir;

 {
   int neighb;

    switch(dir)
     {
      case LEFT: if(( peid % mesh_size) != 0)
                 return peid-1;
                 else return -1;
                 break;
      case RIGHT: if((( peid+1) % mesh_size) != 0)
                 return peid+1;
                 else return -1;
                 break;
      case ABOVE: if( peid >= mesh_size)
                 return peid - mesh_size;
                 else return -1;
                 break;
      case BELOW: if( peid < mesh_size*(mesh_size-1))
                 return peid + mesh_size;
                 else return -1;
                 break;
      }
  }
```

'tree.c'

```
#include "structure.h"

int height;
int proc_no;

structmain()
{
int peid,left_child,right_child,parent,leaf;


                                /* Input interface */
input(height)
no_procs = power(2,height) -1;
initialize();

                                /* Topology Setup */
for(peid = 0; peid < no_procs; peid++)
 {
    if(( parent = get_parent(peid)) != -1)
    connect(peid,parent);
    if((left_child = get_left_child(peid)) != -1)
    connect(peid,left_child);
    if((right_child = get_right_child(peid)) != -1)
    connect(peid,right_child);
 }


start_sim(); /* Set up the simulation environment */
main();      /* Entry point of user programme */
clean();       /* Remove communication links */
terminate();  /* Terminate process 0 */
}


/* Process Creation */
lfork(levell)
int levell;

{
   int np,npl,k,p,pl;
    np = power(2,levell);
    npl = np -1;
    if (get_nodeid() != 0)
     return;
    for(p = 0; p < np; p++) /* Create processes at
                                level levell */

    {
      k = p+npl;
      pfork(k);
      pl = getpid();
      if(pl == child_pid)
       {
           proc_no = k;
           return ;
       }
    }
```

```
                    /* Interprocess Communication */
int send_to_parent(src,dataptr,len)
int src;
char *dataptr;
int len;

  {
    int dest;

    dest = get_parent(src);
    write_to(dest,dataptr,len);
  }

int recv_from_parent(dest,dataptr,len)
int dest;
char *dataptr;
int *len;

  {
    int src;

    src = get_parent(dest);
    write_to(src,dataptr,len);
  }


int send_to_leftchild(src,dataptr,len)
int src;
char *dataptr;
int len;

  {
    int dest;

    dest = get_left_child(src);
    write_to(dest,dataptr,len);
  }

int recv_from_leftchild(dest,dataptr,len)
int dest;
char *dataptr;
int *len;

  {
    int src;

    src = get_left_child(dest);
    read_from(src,dataptr,len);
  }


int send_to_rightchild(src,dataptr,len)
int src;
char *dataptr;
int len;
```

```
   {
     int dest;

     dest = get_right_child(src);
     write_to(dest,dataptr,len);
   }

 int recv_from_rightchild(dest,dataptr,len)
 int dest;
 char *dataptr;
 int *len;

   {
     int src;

     src = get_right_child(dest);
     read_from(src,dataptr,len);
   }


                                  /* Support Routines */
get_height()

{
 return height;
}


 get_noprocs()

  {
   return no_procs;
  }

 get_nodeid()

 {
   return proc_no;
 }


 int get_parent(peid)
 int peid;

 {
  int parent;

    if(peid == 0)
        return -1;

    else {
        if(peid % 2 == 0)
         parent = peid/2 -1;
         else parent = peid/2;
         return parent;
          }
    }
```

'br_cube.c'

```
#include "structure.h"

int dim,drop;
int proc_no;

structmain()
 {
  int peid,id,il;
  char str[LENGTH];
                                /* Input interface */
  input(dim)
  input(drop)

  no_procs = power(2,dim);
  initialize();
                                /* Topology Setup */
 for(peid = 0; peid < no_procs; peid++)
    {
     for(d = 0; d < dim; d++)
       {
        for(p = 0; p < drop; p++)
          {
            ch = get_channel(peid,d);
            if(get_digit(peid,drop,dim,d) != p)
            broad_link(peid,replace(peid,drop,dim,d,p),ch);
          }
       }
    }

 start_sim();

 main();
 clean();             /*Delete the communication links*/
 terminate();
 }
                                /* Process Creation */
  lfork(diml)
  int diml;

   {
    int np,k,p,pl;

    np = power(2,diml);
    if (get_nodeid() != 0)
    return;
    for(p = 0; p < np; p++) /* CHANGE*/
      {
       k = p+np;
       pfork(k);
       pl = getpid();
       if(pl == child_pid)
          {
            proc_no = k;
            return ;
          }
      }
```

```
                /* Interprocess Communication */
gsend_to(src,diml,mesg,len)

 int src,diml;
 char *mesg ;
 int len;

  {
    int ch;
    ch = get_channel(src,diml);
    bwrite(ch,mesg,len);
  }


grecv_from(dest,diml,mesg,len)

 int dest,diml  ;
 char *mesg ;
 int *len;

  {
    int ch;
    ch = get_channel(dest,diml);
    bread(ch,mesg,len);
  }


             /* Support Routines */
get_dim()

  {
    return dim;
  }


get_noprocs()

  {
    return no_procs;
  }


get_nodeid()

 {
   return proc_no;
 }

 get_channel(peid,d)
  int peid,d;

  {
   int ch;
   if(dim == 0)
   ch = peid/drop;
   else ch = (peid % power(drop,d)) + (drop*d);
   return ch;
```

To run the program on the simulator user has to do the following things:

Writing the program

User should call only the routines that are available in the concerned structure files in his program. For example if the user wants to run the program on hypercube structure he should refer the file hp_hube.c and can use the rouines that are available in that file. Please refer APPENDIX B for the details of different structures that are already defined. User should see that final result of the program is always computed at processor 0.

Preparation

Copy the following files to your area.

i) ~jshree/arch/proj/cc_hpcube to run the program on hypercube.

ii) ~jshree/arch/proj/cc_mesh to run the program on 2_d mesh

iii) ~jshree/arch/proj/cc_mms to run the program on MMS

iv) ~jshree/arch/proj/cc_tree to run the program on tree

v) ~jshree/arch/proj/cc_brhpcube to run the program on broadcast hypercube

Compiling the program

To compile the user program he should use concerned compiling command. cc_hpcube, cc_mms, cc_mesh, cc_tree are used to compile the user programmes to simulate hypercube,

MMS, 2_d mesh and tree structures respectively. cc_brhpcube is used to compile the programs written for hypercube with broadcast communication. cc_hpcube,cc_mms, cc_mesh, cc_treeand cc_brhpcube have got the same usage as that of usual c compiler cc.

REFERENCES

[Akl89]
Selim G.Akl, The Design and Analysis of Parallel Algorithms, Prentice-Hall International, 1989.

[Bach86]
Bach M.J., The Design and Implementation of the UNIX Operating System, Prentice Hall of India Ltd., 1986.

[Hwang85]
Hwang K. and Briggs F.A., Computer Architecture and Parallel Processing, McGraw Hill International, 1985.

[Quinn87]
Quinn M., Designing efficient Algorithms for Parallel Computers, McGraw Hill International, 1987.

[Reed87]
Reed. D.A and Fujimoto, R.M., Multicomputer Networks: Message_Based Parallel Processing, MIT Press, Cambridge, Mass., 1987.

[Siegel79]
Howard J Seigel, A model of SIMD Machines and a comparison of various Interconnection Networks, IEEE Transaction on Computers, Vol. C-28 (1979), No.12.

[Stone]
Stone. H.S., High_Performance Computer Architecture, Addison_Wesley Publishing Company, 1988.

[Sun88]
Sun OS User Manual, Sun Microsystems Inc., Mountain View, CA, 1988.